Designing Parallel Operating Systems via Parallel Programming

Eitan Frachtenberg, Kei Davis, Fabrizio Petrini, Juan Fernandez, and
José Carlos Sancho

Los Alamos National Laboratory, Los Alamos, NM 87545, USA,
{`eitanf,kei,fabrizio,juanf,jcsancho`}`@lanl.gov`

Abstract. Ever-increasing demand for computing capability is driving the construction of ever-larger computer clusters, soon to be reaching tens of thousands of processors. Many functionalities of system software have failed to scale accordingly–systems are becoming more complex, less reliable, and less efficient. Our premise is that these deficiencies arise from a lack of global control and coordination of the processing nodes. In practice, current parallel machines are loosely-coupled systems that are used for solving inherently tightly-coupled problems. This paper demonstrates that existing and future systems can be made more scalable by using BSP-like parallel programming principles in the design and implementation of the system software, and by taking full advantage of the latest interconnection network hardware. Moreover, we show that this approach can also yield great improvements in efficiency, reliability, simplicity.

1 Introduction

There is a demonstrable need for a new approach to the design of system software for large clusters. We claim that by using the principles of parallel programming and making effective use of collective communication, great gains may be made in scalability, efficiency, fault tolerance, and reduction in complexity.

Here *system software* refers to all software running on a machine other than user applications. For a workstation or SMP this is just a traditional microprocessor operating system (OS) (e.g. Linux kernel) but for a large, high-performance cluster there are additional components. These include communication libraries (e.g. MPI, OpenMP), parallel file systems, the system monitor/manager, job scheduler, high-performance external network, and more.

Experience with large-scale machines such as Cplant, Virginia Tech's Terascale Cluster, and ASCI's Blue Mountain, White, Q, and Lightning, has shown that managing such machines is time comsuming and expensive. The components of the system software typically introduce redundancies in both functionality (communication and coordination protocols) and in hardware (multiple interconnection networks) and are typically 'bolted together,' each coming from a different developer or vendor, resulting in a multiplication of complexity. Further, for the larger systems efficiency (delivery of theoretical capability), responsiveness, and reliability remain low, indicating that they have already outgrown current, incoherent system software.

We believe that the root of the problem is the use of largely independent, loosely-coupled compute nodes for the solution of problems that are inherently tightly coupled.

A solution to this problem is to better integrate the compute nodes using modern interconnection network hardware. We propose a new methodology for the design of parallel system software based on two cornerstones: 1) BSP-like global control and coordination of *all* of the activities in the machine, and so, 2) treating the system software suite just as any other parallel application. In practice we are able to attain efficient and highly scalable coordination using a very small set of collective communication primitives. From a theoretical point of view this set of primitives should be amenable to formal semantic characterization and analysis (possibly following Lamport [14]), but this remains a potential direction for future research. More practically, this demonstrates new modalities for the use of collective communication.

Much of what we propose here has been implemented and shown to have achieved the desired goals–simplicity, efficiency, effectiveness, and very high scalability. Other components, based on the same primitives, are still in development. However, various subsets of these mechanisms are independent and so may be put into use in the absence of the others, allowing incremental proof of concept.

2 Toward a Parallel Operating System

Distributed and parallel applications (including operating systems) may be distinguished by their use of global and collective operations. Distributed applications typically make much more use of local information and exchange a relatively small number of point-to-point messages. Parallel programs benefit from, and often require, mechanisms for global synchronization and exchange of information, such as barriers, reduction operations, etc. Many OS tasks are inherently collective operations, such as context switching and job launching; other benefit by being cast in terms of collective operations.

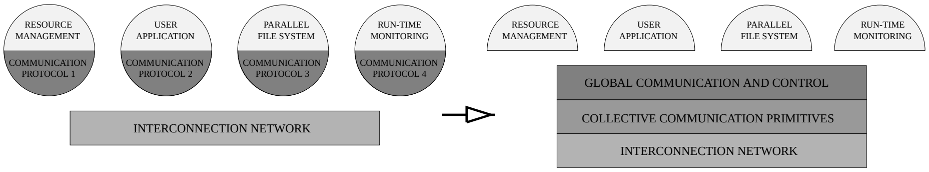
RESOURCE MANAGEMENT
USER APPLICATION
PARALLEL FILE SYSTEM
RUN-TIME MONITORING
COMMUNICATION PROTOCOL 1
COMMUNICATION PROTOCOL 2
COMMUNICATION PROTOCOL 3
COMMUNICATION PROTOCOL 4
INTERCONNECTION NETWORK
RESOURCE MANAGEMENT
USER APPLICATION
PARALLEL FILE SYSTEM
RUN-TIME MONITORING
GLOBAL COMMUNICATION AND CONTROL
COLLECTIVE COMMUNICATION PRIMITIVES
INTERCONNECTION NETWORK

Fig. 1. Communication Protocol

Our vision is that of a cohesive global operating system that is designed using parallel programming paradigms and techniques. Such a unified OS will not only be smaller and simpler than the sum of the parts currently used for system software, but also more efficient because of reduced overhead and redundancy (Fig. 1). All the roles of a cluster OS, such as job launching and scheduling, user-level communication, parallel I/O, transparent fault tolerance, and garbage collection can be implemented over a single communication infrastructure. This layer in turn is designed as a parallel program using the same collective communication primitives.

By using a carefully chosen set of low-latency, scalable communication primitives we can impose a global communication model where both system and user communication is tightly controlled at a fine granularity. The system as a whole behaves as a bulk-synchronous program, where computation and communication are divided into distinct, timed phases. In this model, called Buffered Coscheduling (BCS) [5], all the user and system-level communication is buffered and controlled. The entire cluster marches to the beat of a global strobe that is issued every few hundreds of microseconds. This is somewhat reminiscent of the SIMD model, except the granularity is in terms of time rather than instructions. In the periods between strobes, or *timeslices*, newly-issued communication calls are buffered until the next timeslice. At every strobe, nodes exchange information on pending communication, so that every node has complete knowledge of the required incoming and outgoing communication for the next timeslice. The nodes then proceed to globally schedule those communications that will actually be carried out during the timeslice, and proceed to execute them. Processes that issue non-blocking communication calls are not affected by this model, whereas a blocking call might cause a context switch to the calling process. The advantage of this model is that all the communication is controlled and can be maintained in a known state at every given timeslice, so that problems arising from congestion, out of order arrival, and hot spots can be avoided. As reported in Section 4, these constraints impose little or no overhead for scientific applications, while obtaining the advantages of a more deterministic, controllable machine.

3 Core Primitives and Operating System Functions

Suggested Primitives Our suggested support layer consists of three network primitives. These may be implemented in hardware, software, or a combination of both; commodity hardware exists today that can implement them directly. While this set of primitives is complete (in terms of providing the needed functionality), other primitives might conceivably provide equivalent (or even more) functionality and efficiency.

XFER-AND-SIGNAL Transfer (PUT) a block of data from local memory to a commonly registered memory area, or *global memory*, of a set of nodes, and optionally signal a local and/or a remote event upon completion. Depending on implementation, global data may reside in main or network-interface memory.

TEST-EVENT Check if a local event has been signaled. Optionally, block until it has.

COMPARE-AND-WRITE Arithmetically compare a global variable on a set of nodes to a local value. If the condition is true on *all* nodes, then (optionally) assign a new value to a (possibly different) global variable.

Note that XFER-AND-SIGNAL and COMPARE-AND-WRITE are both atomic operations. That is, XFER-AND-SIGNAL either PUTs data to *all* nodes in the destination set or (in case of a network error) *none* of the nodes. The same condition holds for COMPARE-AND-WRITE when it writes a value to a global variable. Furthermore, if multiple nodes simultaneously initiate COMPARE-AND-WRITEs with identical parameters except for the value to write, then, when all the operations have completed, all nodes will see the same value in the global variable. In other words, both primitives are *sequentially consistent* operations [14]. Test-Event and Compare-And-Write are traditional blocking operations, while Xfer-And-Signal is non-blocking. The only way to check for completion is to Test-Event on a local event triggered by Xfer-And-Signal. These semantics do not dictate whether the mechanisms are implemented by the host CPU or by a network co-processor. Nor do they require that Test-Event yield the CPU, though it may be advantageous to do so.

System Software Requirements and Solutions In this section we discuss how the various responsibilities of a global OS can be realized with the aforementioned mechanisms and communication layer. These OS functions are all discussed and expressed in terms of the proposed abstract layer, showing its sufficiency and completeness for a global OS. Table 1 summarizes these arguments.

Table 1. Network mechanisms usage

Characteristic	Function	Solution
Job Launching	Data dissemination	Xfer-And-Signal
	Flow control	Compare-And-Write
	Termination detection	Compare-And-Write
Job Scheduling	Heartbeat	Xfer-And-Signal
	Context switch responsiveness	Prioritized messages
Communication	Put	Xfer-And-Signal
	Get	Xfer-And-Signal
	Barrier	Compare-And-Write
	Broadcast	Compare-And-Write + Xfer-And-Signal
Storage	Metadata/file data transfer	Xfer-And-Signal
Debugging	Debug data transfer	Xfer-And-Signal
	Debug synchronization	Compare-And-Write
Fault Tolerance	Fault detection	Compare-And-Write
	Checkpointing synchronization	Compare-And-Write
	Checkpointing data transfer	Xfer-And-signal
Garbage Collection	Live state synchronization	Determinism and Compare-And-Write

Job Launching The traditional approach to job launching, including the distribution of executable and data files to cluster nodes, is a simple extension of single-node job launching: data is transmitted using network file systems such as NFS, and jobs are launched with scripts or simple utilities such as rsh or mpirun. These methods do not scale to large machines where the load on the network file system, and the time it would take to serially launch a binary on many nodes, make them inefficient and impractical. Several solutions have been proposed for this problem, all based on software tricks to reduce the distribution time. For example, Cplant and BProc use their own tree-based algorithms to distribute data with latencies that are logarithmic in the number of nodes [2,11]. While more portable than relying on hardware support, these solutions are significantly slower and not always simple to implement [9].

Decomposing job launching into simpler sub-tasks shows that it need only require modest effort to make the process efficient and scalable:

- Executable and data distribution are no more than a multicast of packets from a file server to a set of nodes, and can be implemented using XFER-AND-SIGNAL.
- Launching of a job can be achieved simply and efficiently by multicasting a control message to the target nodes using XFER-AND-SIGNAL. The system software on each node would then fork the process and wait for its termination.
- The reporting of job termination can incur much overhead if each node sends a single message for every process that terminates. This problem can be solved by ensuring that all the processes of a job reach a common synchronization point upon termination (using COMPARE-AND-WRITE) before delivering a single message to the resource manager.

Job Scheduling. Interactive response times from a scheduler are required to make a large machine as usable as a workstation. This in turn implies that the system must be able to perform preemptive context switching with the same latencies we have come to expect from uniprocessor systems, that is, on the order of a few milliseconds. Such latencies are almost impossible to achieve without scalable collective operations: the time required to coordinate a context switch over thousands of nodes can be prohibitively large when using point-to-point messages [12]. Even though the system is able to efficiently context switch between different jobs, concurrent (uncoordinated) application traffic and synchronization messages in the network can unacceptably delay response to the latter. If this occurs even on a single node for even just a few milliseconds it may have a severe detrimental effect on the responsiveness of the entire system [17].

Many contemporary networks offer some capabilities to the software scheduler to prevent these delays. The ability to maintain multiple communication contexts alive in the network securely and reliably, without kernel intervention, is already implemented in some state-of-the-art networks such as QsNet. Job context switching can be easily achieved by simply multicasting a control message or *heartbeat* to all the nodes in the network using XFER-AND-SIGNAL. Our communication layer can guarantee that heartbeats and other system messages get priority over user communication to avoid synchronization problems.

Communication. Most of MPI's, TCP/IP's, and other communication protocols' services can be reduced to a rather basic set of communication primitives, e.g. point-to-point synchronous and asynchronous messages, multicasts, and reductions. If the underlying primitives and protocols are implemented efficiently, scalably, and reliably by the hardware and cluster OS, respectively, the higher level protocols can inherit the same properties. In many cases, this reduction is very simple and can eliminate the need for many of the implementation quirks of protocols that need to run on disparate network hardware. Issues such as flow control, congestion avoidance, quality of service, and prioritization of messages are handled transparently by a single communication layer for all the user and system needs.

Determinism and fault tolerance. When the system globally coordinates all the application processes, parallel jobs can be made to evolve in a controlled manner. Global coordination can be easily implemented with XFER-AND-SIGNAL, and can be used to perform global scheduling of all the system resources. Determinism can be enforced by taking the same scheduling decisions between different executions. At the same time, global coordination of all the system activities helps to identify the states along the program execution at which it is safe to checkpoint.

The tight control of global communication and the induced determinism that follows from this constraint allows for a seamless inclusion of various other important OS services and functionalities. One example is parallel I/O, which can benefit from the hot-spot and congestion avoidance of this model, since all the I/O operations can be scheduled as low-priority communications. The ability to synchronize an entire application to a known state at fine granularity (without having messages en route) is very important for performing global garbage collection, by keeping track of the live state of global objects [13]. Even more important is the ability to use these known states for automatic, fine-grained, coordinated checkpointing. Because of the frequency with which components can fail, one of the main challenges in using large-scale clusters is achieving fault tolerance. The difficulty in using checkpointing techniques for these clusters arises from the quantity of unknown system state at any given point in time, due largely to non-determinism in the communication layer. Eliminating and controlling most of these unknowns allows significant simplification of the implementation of automatic checkpointing and restart at a granularity of a few seconds, far more responsively than current solutions. Furthermore, the handling of the checkpointing traffic is handled and scheduled by the communication layer together with all other traffic, again mitigating flow-control and non-determinism issues.

Implementation and Portability The three primitives presented above assume that the network hardware enables efficient implementation of a commonly registered memory area. Such functionality is provided by several state-of-the-art networks such as QsNet and Infiniband and has been extensively studied [15,16]. We note that some or all of the primitives have have already been implemented in several other interconnects; their expected performance is shown in Table 2. They were originally designed to improve the communication performance of user applications; to the best of our knowledge their usage as an infrastructure for system software has not been explored before this work.

Hardware support for multicast messages sent with XFER-AND-SIGNAL is needed to guarantee scalability for large-scale systems. Software approaches do not scale well to thousands of nodes. In our case, QsNet provides PUT/GET operations, making the implementation of XFER-AND-SIGNAL straightforward.

COMPARE-AND-WRITE assumes that the network is able to return a single value to the calling process regardless of the number of queried nodes. Again, QsNet provides a global query operation that allows direct implementation of COMPARE-AND-WRITE.

4 Results

We have implemented a research prototype, called STORM [9], as a proof of concept of our approach. STORM is a full-fledged resource manager that provides job launching,

Table 2. Measured/expected performance of the core mechanisms for n nodes

Network	Comparison (μs)	Multicast (MB/s)
Gigabit Ethernet [19]	$46 \log n$	Not available
Myrinet [1,3,4]	$20 \log n$	$\sim 15n$
Infiniband [15]	$20 \log n$	Not available
QsNet ([16])	< 10	$> 150n$
BlueGene/L [10]	< 2	$700n$

resource allocation and monitoring, load balancing, and various job scheduling algorithms including space-shared, time-shared, and backfilling variants. STORM's performance has been evaluated, modeled, and compared to several others from the literature [9]. STORM is an order of magnitude faster than the best reported results for job launching, and delivers two orders of magnitude better performance for context switching. STORM was later used to implement several existing and novel scheduling algorithms. In a comprehensive experimental evaluation [7,8], our new algorithms improved the system's utilization and response times both for simple and dynamic workloads. By using our primitives for global resource coordination, the algorithms were better suited to avoiding and mitigating problems of internal and external fragmentation.

We have also implemented BCS-MPI, an MPI library based on the BCS model, on top of STORM. The novelty of BCS-MPI is its use of global coordination of a large number of communicating processes rather than an emphasis on the traditional optimization of the point-to-point performance. Several experimental results [6], using a set of real-world scientific applications, show that BCS-MPI is only marginally slower (less than 10%) than production-grade MPI implementations, but is much simpler to implement, debug, and reason about. We have also obtained some insights and performance numbers for scientific applications providing strong evidence for the feasibility of our approach for transparent fault tolerance [18].

5 Conclusions

We have shown that a BSP-like approach to the design of system software is not only feasible but promises much-needed improvements in efficiency, simplicity, and scalability for all of the key functionalities of a cluster OS. Further, it provides a framework in which effective fault tolerance may be achieved. All of the functions may be implemented in terms of simple collective communication primitives directly supported by currently available interconnection networks.

Concretely, the resource manager STORM implements job launching, resource allocation and monitoring, job scheduling, and load balancing. BCS-MPI implements a high-level communication protocol. The full implementation of the fault tolerance and parallel I/O mechanisms is underway; experimental results provide ample evidence that the desired functionalities and behaviors are achievable.

References

1. R. A. Bhoedjang, T. Rühl, and H. E. Bal. Efficient multicast on Myrinet using link-level flow control. In *International Conference on Parallel Processing*, 1998.
2. R. Brightwell and L. A. Fisk. Scalable parallel application launch on Cplant. In *IEEE/ACM Supercomputing*, 2001.
3. D. Buntinas, D. Panda, J. Duato, and P. Sadayappan. Broadcast/multicast over Myrinet using NIC-assisted multidestination messages. In *Workshop on Communication, Architecture, and Applications for Network-Based Parallel Computing (CANPC '00), High Performance Computer Architecture (HPCA-6) Conference*, Toulouse, France, January 2000.
4. D. Buntinas, D. Panda, and W. Gropp. NIC-based atomic operations on Myrinet/GM. In *SAN-1 Workshop, High Performance Computer Architecture Conference*, 2002.
5. Fabrizio Petrini and Wu-chun Feng. Buffered Coscheduling: A New Methodology for Multitasking Parallel Jobs on Distributed Systems. In *Proceedings of the International Parallel and Distributed Processing Symposium, IPDPS2000*, volume 16, May 2000.
6. J. Fernandez, F. Petrini, and E. Frachtenberg. BCS MPI: A New Approach in the System Software Design for Large-Scale Parallel Computers. In *IEEE/ACM Supercomputing*, 2003.
7. E. Frachtenberg, D. Feitelson, F. Petrini, and J. Fernandez. Flexible CoScheduling: Mitigating Load Imbalance and Improving Utilization of Heterogeneous Resources. In *Proceedings of the International Parallel and Distributed Processing Symposium (IPDPS03)*, 2003.
8. E. Frachtenberg, D. G. Feitelson, J. Fernandez, and F. Petrini. Parallel Job Scheduling under Dynamic Workloads. In D. G. Feitelson, L. Rudolph, and U. Schwiegelshohn, editors, *Job Scheduling Strategies for Parallel Processing*, volume 2862 of *Lect. Notes Comput. Sci.*, pages 208–227. Springer Verlag, 2003.
9. E. Frachtenberg, F. Petrini, J. Fernandez, S. Pakin, and S. Coll. STORM: Lightning-Fast Resource Management. In *Proceedings of IEEE/ACM Supercomputing*, 2002.
10. M. Gupta. Challenges in developing scalable scalable software for bluegene/l. In *Scaling to New Heights Workshop*, Pittsburgh, PA, May 2002.
11. E. Hendriks. BProc: The Beowulf distributed process space. In *Proceedings of the* 16th *Annual ACM International Conference on Supercomputing*, 2002.
12. A. Hori, H. Tezuka, and Y. Ishikawa. Overhead analysis of preemptive gang scheduling. In D. G. Feitelson and L. Rudolph, editors, *Job Scheduling Strategies for Parallel Processing*, pages 217–230. Springer Verlag, 1998.
13. T. Kamada, S. Matsuoka, and A. Yonezawa. Efficient parallel global garbage collection on massively parallel computers. In G. M. Johnson, editor, *Proceedings of IEEE/ACM Supercomputing 1994 (SC'94)*, pages 79–88, 1994.
14. L. Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Transactions on Computers*, C-28(9):690–691, Sept 1979.
15. J. Liu, J. Wu, D. K. Panda, and C. Shamir. Designing clusters with Infiniband: Early experience with Mellanox technology. Submitted for publication.
16. F. Petrini, W. Feng, A. Hoisie, S. Coll, and E. Frachtenberg. The Quadrics Network: High-Performance Clustering Technology. *IEEE Micro*, 22(1):46–57, Jan/Feb 2002.
17. F. Petrini, D. Kerbyson, and S. Pakin. The Case of the Missing Supercomputer Performance: Achieving Optimal Performance on the 8,192 Processors of ASCI Q. In *Proceedings of IEEE/ACM Supercomputing 2003 (SC'03)*, Phoenix, AZ, Nov 2003.
18. J. C. Sancho, F. Petrini, G. Johnson, J. Fernández, and E. Frachtenberg. On the Feasibility of Incremental Checkpointing for Scientific Computing. In *Proceedings of the 18th International Parallel & Distributed Processing Symposium*, Apr 26–30, 2004.
19. P. Shivam, P. Wyckoff, and D. Panda. EMP: Zero-copy os-bypass nic-driven gigabit ethernet message passing. In *Proceedings of IEEE/ACM Supercomputing)*, 2001.